Normalization

Table of Contents

[Overview 2](#_Toc115682784)

[Anomalies in DBMS 2](#_Toc115682785)

[Normalization Forms 3](#_Toc115682786)

[First normal form (1NF) 3](#_Toc115682787)

[Second normal form (2NF) 5](#_Toc115682788)

[Third Normal form (3NF) 8](#_Toc115682789)

# Overview

Normalization is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. Let’s discuss about anomalies first then we will discuss normal forms with examples.

## Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – **Insertion, update and deletion anomaly**. Let’s take an example to understand this.

Example: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp\_id, emp\_name, emp\_address and emp\_dept for storing employee’s id, name, address and department in which the employee works. At some point of time the table looks like this:

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_address | emp\_dept |
| 101 | Jayesh | Delhi | D001 |
| 101 | Jayesh | Delhi | D002 |
| 123 | Megha | Agra | D890 |
| 166 | Glenn | Chennai | D900 |
| 166 | Glenn | Chennai | D004 |

The above table is not normalized. We will see the problems that we face when a table is not normalized.

**Update anomaly:** In the above table we have two rows for employee Jayesh as he belongs to two departments of the company. If we want to update the address of Jayesh then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Jayesh would be having two different addresses, which is not correct and would lead to inconsistent data.

**Insert anomaly:** Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp\_dept field doesn’t allow nulls.

**Delete anomaly:** Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp\_dept as D890 would also delete the information of employee Megha since she is assigned only to this department.

To overcome these anomalies we need to normalize the data. In the next section we will discuss about normalization.

# Normalization Forms

Here are the most commonly used normal forms:

* First normal form(1NF)
* Second normal form(2NF)
* Third normal form(3NF)

## First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_address | emp\_mobile |
| 101 | Radhika | New Delhi | 8912312390 |
| 102 | Amit | Kanpur | 8812121212,  9900012222 |
| 103 | Rohit | Chennai | 7778881212 |
| 104 | Veer | Bangalore | 9990000123,  8123450987 |

Two employees (Amit & Veer) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is not in 1NF as the rule says “**each attribute of a table must have atomic (single) values**”, the emp\_mobile values for employees Amit & Veer violates that rule.

To make the table complies with 1NF we should have the data like this:

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_address | emp\_mobile |
| 101 | Radhika | New Delhi | 8912312390 |
| 102 | Amit | Kanpur | 8812121212 |
| 102 | Amit | Kanpur | 9900012222 |
| 103 | Rohit | Chennai | 7778881212 |
| 104 | Veer | Bangalore | 9990000123 |
| 104 | Veer | Bangalore | 8123450987 |

## 

## Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

* Table is in 1NF (First normal form)
* No **non-prime attribute** is dependent on the proper subset of any **candidate key** of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

|  |  |  |
| --- | --- | --- |
| teacher\_id | Subject | teacher\_age |
| 111 | Maths | 38 |
| 111 | Physics | 38 |
| 222 | Biology | 38 |
| 333 | Physics | 40 |
| 333 | Chemistry | 40 |

Candidate Keys: {teacher\_id, subject}

Non prime attribute: teacher\_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute **teacher\_age** is dependent on teacher\_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says “no non-prime attribute is dependent on the proper subset of any candidate key of the table”.

To make the table complies with 2NF we can break it in two tables like this:

teacher\_details table:

|  |  |
| --- | --- |
| teacher\_id | teacher\_age |
| 111 | 38 |
| 222 | 38 |
| 333 | 40 |

teacher\_subject table:

|  |  |
| --- | --- |
| teacher\_id | Subject |
| 111 | Maths |
| 111 | Physics |
| 222 | Biology |
| 333 | Physics |
| 333 | Chemistry |

Now the tables comply with Second normal form (2NF).

## Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

* Table must be in 2NF
* Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

* X is a [super key](https://beginnersbook.com/2015/04/super-key-in-dbms/) of table
* Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee\_details that looks like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_zip | emp\_state | emp\_city | emp\_district |
| 1001 | Durgesh | 282005 | UP | Agra | Dayal Bagh |
| 1002 | Ajeet | 282005 | UP | Agra | Dayal Bagh |
| 1006 | Lora | 282007 | TN | Chennai | Urrapakkam |
| 1101 | Lilly | 292008 | UK | Pauri | Bhagwan |
| 1201 | Steve | 222999 | MP | Gwalior | Ratan |

**Super keys**: {emp\_id}, {emp\_id, emp\_name}, {emp\_id, emp\_name, emp\_zip}…so on

**Candidate Keys**: {emp\_id}

* Non-prime attributes: all attributes except emp\_id are non-prime as they are not part of any candidate keys.
* Here, emp\_state, emp\_city & emp\_district dependent on emp\_zip. And, emp\_zip is dependent on emp\_id that makes non-prime attributes (emp\_state, emp\_city & emp\_district) transitively dependent on super key (emp\_id). This violates the rule of 3NF.
* To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

employee table:

|  |  |  |
| --- | --- | --- |
| emp\_id | emp\_name | emp\_zip |
| 1001 | Durgesh | 282005 |
| 1002 | Ajeet | 282005 |
| 1006 | Lora | 282007 |
| 1101 | Lilly | 292008 |
| 1201 | Steve | 222999 |

employee\_zip table:

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_zip | emp\_state | emp\_city | emp\_district |
| 282005 | UP | Agra | Dayal Bagh |
| 222008 | TN | Chennai | M-City |
| 282007 | TN | Chennai | Urrapakkam |
| 292008 | UK | Pauri | Bhagwan |
| 222999 | MP | Gwalior | Ratan |